



# Universiteit Leiden

## Opleiding Informatica

Examining out of Bounds Defense Systems'  
Performance Against Independent CVEs

Name:	Elgar R. van der Zande
Studentnr:	s1485873
Date:	June 19, 2019
1st supervisor:	Dr. E. van der Kouwe
2nd supervisor:	Dr. K.F.D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)  
Leiden University  
Niels Bohrweg 1  
2333 CA Leiden  
The Netherlands

## **Abstract**

In this thesis, we present a novel framework that can be used to automatically test C/C++ defense systems against an independent set of CVEs. The framework is capable of generating an HTML report that clearly shows the violation detection performance of the defense systems, as it contains a table that shows for each defense system which CVEs it can detect. To demonstrate the capabilities of our framework, we have included several defense systems and CVEs. In this process, we restricted ourselves to CVEs that contain an out of bounds memory violation. Subsequently, we evaluate the performance of the different defense systems based on the generated report. Moreover, we also show how to add new defense systems and CVEs to the framework, allowing the user to compare new software against the already existing set of defense systems and CVEs.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Contributions . . . . .	5
1.2	This thesis is organized as follows . . . . .	5
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Defense systems . . . . .	6
2.2	CVE Database . . . . .	7
<b>3</b>	<b>Related Work</b>	<b>9</b>
3.1	Benchmark . . . . .	9
3.2	Theoretical comparision . . . . .	10
<b>4</b>	<b>Overview</b>	<b>12</b>
<b>5</b>	<b>Design</b>	<b>14</b>
5.1	Setup . . . . .	14
5.2	Running Tests . . . . .	15
5.3	Detection of Violation Types . . . . .	15
5.4	Adding/Changing content . . . . .	17
5.5	Data Representation . . . . .	18
<b>6</b>	<b>Implementation</b>	<b>19</b>
6.1	Source Files . . . . .	20
6.2	Adding a Defense System to the Framework . . . . .	21
6.2.1	Address Sanitizer . . . . .	23
6.2.2	LowFat . . . . .	23
6.2.3	SafeCode . . . . .	23
6.2.4	Softbound . . . . .	23
6.2.5	Failed Defense Systems . . . . .	24
6.3	CVE definition . . . . .	24
6.3.1	CVE-2014-0160 . . . . .	25
6.3.2	CVE-2017-8361 . . . . .	26
6.3.3	CVE-2017-8364 . . . . .	27
6.3.4	CVE-2017-9048 . . . . .	27
6.3.5	CVE-2017-9929 . . . . .	27

6.3.6	CVE-2017-9928 . . . . .	27
6.3.7	CVE-2018-16375 . . . . .	27
6.3.8	CVE-2018-7487 . . . . .	28
6.3.9	CVE-2018-7648 . . . . .	29
6.3.10	CVE-2018-8905 . . . . .	29
<b>7</b>	<b>Evaluation</b>	<b>30</b>
7.1	Address Sanitizer . . . . .	30
7.2	LowFat . . . . .	31
7.3	SafeCode . . . . .	32
7.4	Softbound . . . . .	32
<b>8</b>	<b>Discussion</b>	<b>33</b>
8.1	Generated Results . . . . .	33
8.2	Technical Shortcomings and Further Work . . . . .	34
<b>A</b>	<b>Using the Framework</b>	<b>37</b>
A.1	Building the Defense Systems . . . . .	37
A.2	Running Tests . . . . .	37
A.3	Generating a Report . . . . .	38
A.4	Cleaning up . . . . .	39
<b>B</b>	<b>Safecode Patch</b>	<b>40</b>
<b>C</b>	<b>Softbound Patch</b>	<b>42</b>

# Chapter 1

## Introduction

C and C++, despite their notoriously easy introduction of security vulnerabilities, remain among the most popular programming languages to date, the reason being speed and flexibility. However, this reason is precisely what causes the vulnerabilities in the first place. For example, not checking the bound of an array when dereferencing its memory pointer can lead to an out of bounds (OOB) memory access, resulting in undefined behavior and in some cases this behavior can be used to cause a security breach.

There are different ways a vulnerability can potentially be exploited, ranging from denial of service (DoS) to remote code execution<sup>1</sup> or leaking of data (e.g., Heartbleed). Preferably, one wants to prevent the latter ones in favor of the first, and this is where a defense system comes into play. Its job is to terminate a program prematurely when a violation occurs such that an exploit cannot do any damage. Generally speaking, to activate a defense system the application needs to be recompiled with the appropriate compiler for the instrumentation instructions to be built into the binary. However, there are also sanitizers available that do not require recompilation of the application (for example Valgrind [17]).

The most widely adopted sanitizer is ASan [21] because of its high true positive rate. However, this does come with a cost, namely the runtime overhead of the application is up to 3.8 times [18]. To decrease the runtime overhead of a defense system numerous other defense systems have been created, concerning mainly production systems instead of just focusing on debugging. These defense systems aim to have a lower runtime overhead by sacrificing their detection rate slightly. It is therefore interesting to compare different defense systems and check whether they satisfy their specifications when tested against independent vulnerabilities chosen from the Common Vulnerabilities and Exposures (CVE) database [14].

Our thesis discusses the comparison of different defense systems, albeit limited to spacial memory errors (i.e., OOB) only. The tested defense systems are: ASan (for both GCC, and Clang), LowFat [7] [9] [8], Safecode [4], and Softbound [16]. We test these defense systems against a number of (recent) CVEs, using our automated framework [25] such that minimal effort is required. Finally, the framework generates an HTML report that provides easy insight into whether a defense can detect a certain vulnerability. The output of (failed) tests can also be easily be looked up in the report providing the user with more information. This feature is especially useful when a compiler fails to build a source, as the source of the problem can then quickly be looked up.

---

<sup>1</sup>Although this is becoming more and more difficult to pull off, because of hardening techniques built into recent kernels.

Conclusions that we draw from the report are: how the different defense systems compare to each other; what is the detection rate of a defense system; how compatible are they with existing source code, as this compatibility is often claimed to be a feature of a defense system (i.e., they are a drop in replacement of an ordinary compiler). Moreover, the comparison is especially interesting since not much research is available that compares defense systems in such a way.

## 1.1 Contributions

- A novel framework for testing defense systems.
- Interpretation of generated results.
- Collection of CVEs ready to be used, with the framework.
- Collection defense systems ready to be used.
- Review of several defense systems.

## 1.2 This thesis is organized as follows

We start by introducing the concepts required to understand the material. We then compare our research to other work published in this area in our Related Work. Followed by a brief introduction of our framework in the Overview of this thesis.

In Design, we explain the design choices we made in order to make the setup of the framework as simple as possible. Furthermore, we discuss what choices we made in running tests, and how this influences the process of adding more CVEs or defense systems. We conclude the chapter with an explanation of the choices we made to represent the data in a clear manner.

The following chapter is Implementation, here we discuss how we implemented our design decisions. Here we also discuss changes we made to the defense systems we integrated into the framework; this includes both an explanation of the compile process, and how the defense systems should be used to compile the source code of a program. We end the chapter with an overview of every CVE in the framework.

In the Evaluation, we evaluate the defense systems, based on the generated results. moving on to the Discussion, we reason about the capabilities of the framework, and how well we succeeded in meeting our design plan. In the second section of the Discussion, we present a number of items that can be improved to overcome shortcomings or can be added as future work.

Finally, in the appendices, we give a comprehensive explanation of how to build the framework and how to use it. The final sections contain patch files of defense systems we applied.

# Background

## 2.1 Defense systems

A defense system is an application that detects faults in (C/C++) programs or libraries. Generally speaking, a defense system does not catch all types of vulnerabilities but instead focusses only on a small subset of flaws. In this thesis, we focus on a specific subset of vulnerabilities namely memory safety violations. These violations happen when an application accesses parts of the memory that are not (or no longer) valid. Such invalid accesses can happen because of numerous reasons, for instance, a wrongly calculated size of a buffer resulting in a un/overflow. It is the job of the defense system to catch such a violation, print a (preferably) detailed description of what has happened and then terminate the program. This way it is not possible to exploit the vulnerability. Evidently, exploitation of an application only occurs when it is running in a production stage (i.e. not when developing/debugging), a defense system that runs on such a production system is generally referred to as an exploit mitigation system, whereas a system that is intended to help with debugging of a program we refer to as a sanitizer [22].

For both systems, there are some contradicting design specifications, and therefore the two types of systems are generally not interchangeable. Essentially a sanitizer focuses on an accurate description of the violation, thus making debugging a more straightforward process, whereas the exploit mitigation system provides a less detailed information message in favor of lower runtime overhead. For example, taking advantage of aligned memory locations can reduce the time it takes to do bounds check, but it may reduce accuracy, as the alignment forces the object size to be a power of 2.

Another example of checks that a defense sometimes omits or are infeasible to check, depending on the method used for tracking memory objects, is the violation of interior memory locations in objects (e.g., member variables of a `struct`). As these are guaranteed to be stored successively in memory (i.e., there can be no objects in between) it may be impossible to add instrumentation for interior objects. Even if the defense system uses a per object tracking method it may still be impossible to correctly detect every memory violation, consider the following code listing:

```

struct foo
{
    uint32_t x;
    uint32_t y;
};

int main()
{
    struct foo bar;

    memset(&bar, 0, sizeof(struct foo));
    memset(&bar.x, 0, sizeof(struct foo));
}

```

The first `memset` is a valid operation, however the second `memset` overwrites the `y` object from the `struct foo` object. Although these two lines seem different, internally, the calls are equivalent because the first member variable has the same location as the beginning of the `struct`. This example shows the difficulties of writing a defense system, especially when considering the constraint that the application binary interface (ABI) should not change<sup>1</sup> as this would rule out the use of proprietary closed source software (libraries).

Roughly speaking the following two distinctions can be made in memory safety violations: temporal violations and spacial violations [24]. These violations happen when data is read or written to a memory location that is not intended to be accessed. One way this can happen is when an array is accessed with the wrong index, as this results in dereferencing a memory location that is not part of the array, resulting in a memory violation. This type of memory error (i.e., reading/writing beyond the bounds of an object) is called a spatial safety violation; the type of violation is what this thesis focusses on. A different kind of memory error is when memory is accessed that is no longer valid. Such memory access can happen in multiple ways. For example, when dereferencing a pointer after its memory has been freed (use-after-free [22]) or when a pointer is used that pointed to an automatic variable that has gone out of scope (use-after-scope [22]). These kinds of memory violations are referred to as temporal memory violations. However, we do not test them in the framework, due to time constraints, and we mention them only for completeness.

## 2.2 CVE Database

To test the defense systems a set of known vulnerabilities is needed, one way to do this is to create a test set artificially. The benefit of this approach is that the creator has full control over the types of memory errors and how they are triggered (e.g., it becomes trivial to choose the number of bytes to overflow). Also, the readability of the code is very high, as a few lines of code capture a complete vulnerability. Wilander and Kamkar take such an approach for their research [26].

Although this is a valid approach one problem remains, which is that this only gives limited insight into how well a defense system performs against real-world vulnerabilities with all of its complication involved. For instance, the program loses information about its state in external

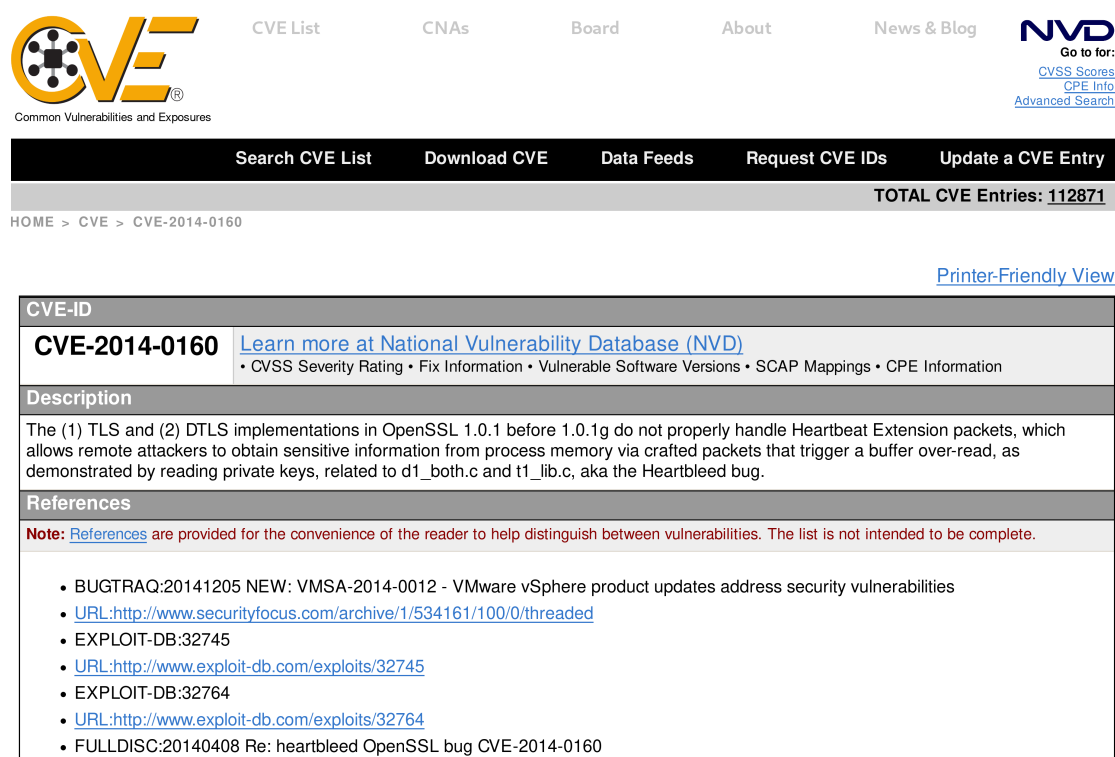
---

<sup>1</sup>This is strictly speaking not a requirement. However, all of the defense systems used here meet this specification.



libraries, as a defense system may not be able to instrument them, and thus the metadata for the defense system cannot be updated. Luckily there exist databases that aim to register vulnerabilities in (open source) code, the Common Vulnerabilities and Exposures (CVE) database being one of them. Incidentally, this is also the database that is used for this research to find the necessary vulnerabilities. As mentioned we only focus on spatial OOB memory errors in open source software. These requirements are needed because all of the defense systems need to be able to recompile the source code to create an instrumented binary.

Figure 2.1 depicts a CVE entry; each entry has a unique ID and a short description. This description, generally speaking, contains the following information. Firstly, the name and version of the affected program/library. Secondly, the nature of the vulnerability (is it a buffer overflow, use after free). Thirdly, a rough location of the vulnerability in the code, usually a function and a filename (at least this is the case for OOB errors in open source projects, where such a remark makes sense). Finally, the malignancy of the vulnerability and possible types of attacks, although this item is often omitted.



The screenshot shows the CVE-2014-0160 entry on the National Vulnerability Database (NVD) website. The page layout includes a header with navigation links (CVE List, CNAs, Board, About, News & Blog) and a search bar. The main content area displays the CVE ID, a link to the NVD, and a detailed description of the vulnerability. The description states that the (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1\_both.c and t1\_lib.c, aka the Heartbleed bug. Below the description, there is a section for references, which includes links to bugtraq, exploit-db, and full-disc, as well as a note about the references.

CVE-ID
<b>CVE-2014-0160</b> <a href="#">Learn more at National Vulnerability Database (NVD)</a> • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information
<b>Description</b> The (1) TLS and (2) DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer over-read, as demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.
<b>References</b> <b>Note:</b> <a href="#">References</a> are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete. <ul style="list-style-type: none"> <li>• BUGTRAQ:20141205 NEW: VMSA-2014-0012 - VMware vSphere product updates address security vulnerabilities</li> <li>• URL:<a href="http://www.securityfocus.com/archive/1/534161/100/0/threaded">http://www.securityfocus.com/archive/1/534161/100/0/threaded</a></li> <li>• EXPLOIT-DB:32745</li> <li>• URL:<a href="http://www.exploit-db.com/exploits/32745">http://www.exploit-db.com/exploits/32745</a></li> <li>• EXPLOIT-DB:32764</li> <li>• URL:<a href="http://www.exploit-db.com/exploits/32764">http://www.exploit-db.com/exploits/32764</a></li> <li>• FULLDISC:20140408 Re: heartbleed OpenSSL bug CVE-2014-0160</li> </ul>

Figure 2.1: CVE entry of the well-known heartbleed vulnerability (2014), a short description contains the affected version, the rough location of the vulnerability and the possible risks of the vulnerability. Below the description some links can be found pointing to other valuable information, e.g., proof of concept (POC) files/methods, patches or (proposed) fixes, etc.

## Related Work

In this area, roughly speaking, the research done divides into two groups, either a purely theoretical comparison or benchmark of multiple defense systems. By purely theoretical we mean that the authors compare the defense systems based on their specifications; however, these specifications are not tested and are assumed to be true. The other category is benchmarks, in which the defense systems run a predefined set of test, and the results are then used to judge the defense systems. Because the benchmark category has the most similarities, we begin by discussing these first.

### 3.1 Benchmark

The only authors that conducted an empirical experiment, discussed in our related work at least, are Wilander and Kamkar in their: “A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention” [26]. Beside the experimental results, they also have a theoretical analysis that they use to compare the results with, the comparison shows a very close resemblance to the real-world experimental results. However, a clear explanation of how they derived the theoretical result is missing.

We now compare our study, starting with the similarities between the experiments. It is clear that both studies carry out a real-world experiment, to verify the claims made in the paper accompanied by the defense system. Furthermore, Wilander and Kamkar also provide all the tools that are used to run the experiments, and therefore the results are easily reproduced. Also, the scale of both experiments is comparable in the sense that Wilander and Kamkar have 20 attacks in their test set, whereas we have 10 CVEs thus the size is in the same order of magnitude. For the defense systems more or less the same applies Wilander and Kamkar test 4 (or 5 if one counts libsafe [2] and libverify [1] separately) publicly available defense systems, this is the same number of defense systems as we currently have in our framework (Counting all occurrences of ASan as one).

However, there also interesting differences: Wilander and Kamkar generate their tests, while we search in the CVE database for suitable memory errors. By creating their tests Wilander and Kamkar gain accurate control over what kind of memory violation is happening, and this allows them to create specific attack types. The result is that they only focus on a specific type of attack, namely they try to change the control flow of the process and use this to measure whether the attack was prevented successfully by the defense system. Our approach is entirely different

in the sense that we have very little control over the type of vulnerability, i.e., we can only choose if we add a CVE or not. However, we do end up with a more broad range of vulnerability types namely all OOB memory errors. A vulnerability type that is present in our set of tests but not in theirs are the vulnerabilities where data is leaked, for example, Heartbleed.

Another noteworthy difference is that our research also focuses on the explanation and working principles of the framework, such that it can easily be reused or extend. Wilander and Kamkar do not discuss their setup in great detail, and there is no mention of how to use their test setup.

In terms of results, we find that there are similarities Wilander and Kamkar also find that the rate of success is far from perfect. Their best defense systems scores around 50% and this is more or less on par with our findings (if one leaves ASan out of the results). Leaving ASan out is valid because there is a bias for ASan detectable memory violations in the CVE database (as discussed in section 7.1). Furthermore, ASan is strictly speaking more of a sanitizer than a defense system, and therefore a higher score is also expected. Another striking difference is the fact that Wilander and Kamkar do provide a theoretical hypothesis to accompany their results; unfortunately, our thesis lacks such a theoretical prediction.

## 3.2 Theoretical comparison

Among the research that is based around theoretical comparison of multiple defense systems are “SoK: Sanitizing for Security” written by Song et al. [22] and “SoK: Eternal War in Memory” written by Szekeres et al. [24] We start with a description of Song et al. and make a comparison to our research where possible. After this comparison, we also briefly discuss Szekeres et al. in the same manner as the previous paper.

The authors of “SoK: Sanitizing for Security” provide a systematic overview of sanitizers and group them by their ability to cover specific vulnerabilities. They do this by starting with an extensive explanation of low-level vulnerabilities among which are memory safety, uninitialized variables, pointer type errors, and variadic function misuse.

After all of the vulnerability types are discussed, the authors present 34 defense systems (or more specifically sanitizers). In their collection of defense systems are both publicly available and commercial tools. They introduce the defense systems via the explanation of various bug-finding techniques (i.e., how a defense system implements its checking routines), per technique the authors mention appropriate defense systems that can be used to counter attacks.

We now compare Song et al. to our research; we start with the similarities. Song et al. also focus on C/C++ defense systems and have common that all of them work at the runtime level. Almost all of the defense systems that we use are also included in the article of Song et al., except for Safecode. The reason that Safecode is left out is that it is no longer actively maintained. Another similar result is that Song et al. find that Softbound has trouble running their benchmarks, these problems are similar to what we find in the sense that Softbound has much trouble with compiling and executing the source codes that we provide.

There are however many differences between our and their research, to start of the aim of the research is different. Song et al. try to convey as much information about the currently relevant sanitizers as possible, whereas we aim to introduce a new framework that allows for easy testing of multiple defense systems. These different approaches logically result in a different outcome. We show the performance number of a defense system whereas Song et al. show the features a defense system supports. Furthermore, the research aim of Song et al. gives rise to a much larger number of defense systems. In conclusion Song et al. provide a complete overview of currently relevant defense systems, whereas we introduce a new framework for testing defense systems’

performance.

We now discuss “Sok: Eternal War in Memory” written by Szekeres et al., once again we first give a summary and then discuss similarities and differences between the paper and our research. The article written by Szekeres et al. is, just like the previous article, a mostly theoretical overview of defense techniques. However, Szekeres et al. describe a broader overview of exploits mitigation techniques, including the ones in our research.

The paper first describes well-known attacks that someone can execute on vulnerable C/C++ programs. Among these attack types are the following: control-flow hijacking, data only attacks and information leakage. The authors then describe numerous ways to counter the attacks mentioned above using different defense techniques. It is important to remember that the countermeasures they describe are not limited to the types of defense systems used in our research. They start by describing multiple techniques to prevent buffer overwrites. They then continue to describe other hardening techniques that aim to make exploitation more difficult like ASLR (address space layout randomization). Finally, they also discuss the use of instrumented binaries. Defense systems just like the ones we use in research, create these binaries.

In conclusion, the research done by Szekeres et al. focusses on a much broader range of attacks and defenses than our research. However, this range of defenses also includes the use of instrumented binaries, like the ones we tested in our research. They show that when an instrumented binary can catch all memory violations, especially when temporal memory violations are also caught, it becomes impossible to execute memory corruption exploits. However, a 100% detection rate is rarely achievable, because of limitations in the technique used to detect memory violations or the use of external libraries that cannot be instrumented. Szekeres et al. do not give an indication about what the performance number of a defense system might be, and therefore it is hard to conclude what the ratio will be at which a defense system will divert an attack; this shows the importance of running real-world tests that our framework provides.

# Chapter 4

## Overview

We created a framework that can be used to automate the testing process of defense systems. The defense systems in the framework are automatically downloaded and built. After this step, each CVE is compiled using every defense system, and the created binary is then used to test if the defense system can detect the vulnerability. This test is done either using a POC file as provided in the CVE, or by a POC file that we created. The results folder stores the results; these are later used to create an HTML report. Using three simple commands, the framework carries out all of the steps mentioned above.

To interact with the framework one tool is used, this tool is called `run.py` and is divided up into six sub-commands. The first three sub-commands build the defense systems on the local machine, run the tests and generate the report. Furthermore, there are also three clean commands for cleaning up the defense systems, test/build files and result files. Figure 4.1 shows a flowchart of the three most essential stages the framework goes through.

For the framework to work correctly, about *30 GiB* of free storage needs to be available in the folder one wants to place the framework in. Most of the space is taken by the defense systems, as their sizes range from *1 GiB* to *10 GiB* once compiled and installed. A much smaller amount of space is used by the CVE vulnerability set, around *3 GiB* once they are built, though this also depends on the defense system that is in use.

The framework creates a result folder, where it stores both test results and log files. The log files are useful for debugging and have the following layout. Lines that start with ‘>>>’ denote a command that is executed by the framework. All of the successive lines contain the output of that command; these include both stdout and stdin. The framework also flushes log files after each line that it writes, making it possible to follow the output real-time (e.g., `tail -f results/some_file.log`) when executing a long-running process.

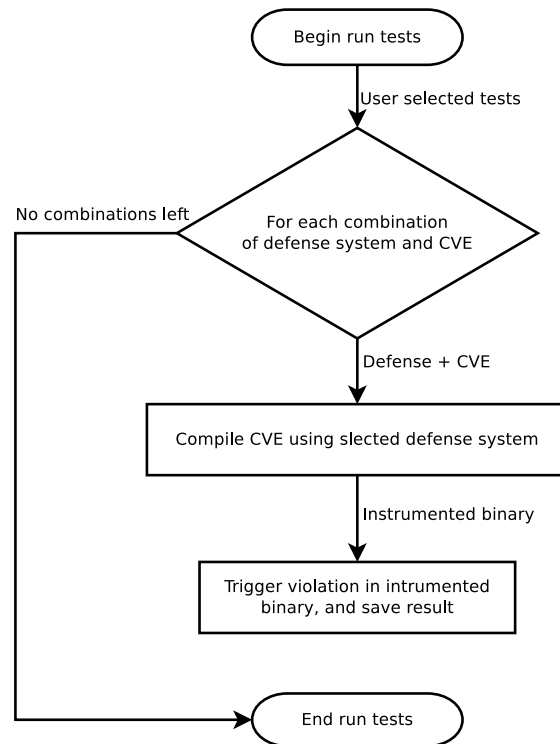


Figure 4.1: The flowchart shows the process the framework goes through when generating the result files. A user can initiate this process using the `run_tests` flag, and specify a subset of the available tests for which to produce the results.

# Chapter 5

## Design

This chapter describes the design choices for the framework and how the framework incorporates these choices. To ease the explanation of the design some definitions will be given first. The set  $C$  consists of the collection of CVEs in the framework, note that when we talk about a ‘CVE in the framework’ we mean the associated vulnerability and the software, i.e., assets (POC files, scripts, etcetera) to trigger this vulnerability. The set  $D$  contains all of the defense systems implemented in the framework. The set  $T = C \times D$  thus contains all possible combinations of CVEs and defense systems; this set is called the set of tests.

The framework is written in Python 3.7 the reason being that it is a powerful language: it allows for easy dynamic loading of classes (this is used to define CVEs and Defenses); has excellent built-in tools for working with JSON; works exceptionally well for parsing texts, and it is an overall well established and accepted language. Another reasonable choice would have been Bash. However, we felt like the project was slightly too big and it required a more powerful language, as such using Bash would result in a convoluted script. The only benefit would have been that adding CVEs and defenses as Bash scripts rather than Python scripts, as Bash is a more natural language for this kind of work.

### 5.1 Setup

It should be easy to set up the framework on a Linux computer, however since compilers are such complex programs, and we are also dealing with aging software, writing a framework that will work out of the box on any OS/Linux distribution is almost infeasible. Therefore, the choice we made was to create an application that works on Ubuntu 18.04 LTS; this is a viable solution since nowadays setting up a VM (virtual machine) is almost effortless. A shell script is provided to prepare the Ubuntu system; this script will install all of the necessary dependencies.

If one still wants to use the framework on a non-supported distribution, the framework itself should run just fine as long as a Python 3.7 interpreter, and the equivalent packages found in the shell script are installed. The automated compilation of the defenses will presumably be more problematic since Ubuntu-specific patches are applied. The easiest solution is to see what works and go with that, as it is not a problem to run the framework with missing defense systems (the report will say ‘build failed’). Alternatively, it is also possible to provide different patch files or compile problematic defense systems by hand. In the latter case, all that needs to be done is to change the `build.params` variable in the concerning defense’ `def.py`.

For the CVE the same thing goes as for the defense systems when running on an unsupported OS. However, the CVE collections consist of projects that compile easily. Therefore, as long as the right dependencies and build systems are installed most of the CVEs in the framework should work just fine.

## 5.2 Running Tests

The primary goal of the framework is to reduce the amount of manual work that needs to be done and reduce the time the user has to wait for the report to be generated. It is evident that the number of tests  $|T| = |C| \cdot |D|$ , indicating that it is not a task that is quickly done by hand, even for small  $|C|$  and  $|D|$ . At this point running the framework on a typical PC already takes more than an hour to complete. The framework should generate these tests and structure their results into an easy to handle file format. Another added benefit of using our framework is that reproducing/rerunning the tests becomes trivial.

To reduce the time it takes to run the tests the framework will only run tests that do not have a result file yet. Meaning that when adding a new CVE or defense system, the framework only runs the newly created tests. However, this approach poses another problem namely that in some cases it should be possible to rerun tests and in particular rerun failed tests. Another useful feature is to be able to reduce the selection of CVEs and defenses, especially when ‘all’ or ‘failed’ tests get rerun. To solve the problems mentioned above the following solution is used. Some optional parameters are added to the `run_tests` subcommand. These parameters are used to change the selection of CVEs and defense systems, allowing the user to omit already built tests.

After the set of tests has been generated and filtered, each test will be executed. The test process consists of the following steps. Opening the appropriate files for logging and writing the results. Loading the CVE and defense system (these are implemented as classes and are therefore dynamically loaded). At this point, the result file is partly being build up (still in memory though) as names, version numbers, and so forth are now known. After this step, the CVE is built and executed. This process happens in 3 separate stages: clean, build and exploit. The first step cleans the directory, leaving only source file archives, POC (proof of concept) files helper scripts, etcetera. After this step the build stage is executed; here the source gets build using the appropriate defense system. Generally speaking, this is done by setting some environment variables to change the compiler and compiler/linker flags. Unfortunately, there is not a single way to do this for all build systems. Therefore, adapting the instructions from the framework to the build system used in a particular open source project is also an essential step in the build phase and is defined in the `def.py` file.

Furthermore, if the project has been compiled and linked successfully, the program gets executed such that it triggers the violation. If the defense system detects the violation, it will print an error message and abort the program. The output of the program is used to determine if the defense succeeded and the framework saves the result in the result file. Moreover, if something fails during the steps mentioned above the entire process is interrupted, in this case, ‘build failed’ is noted in the result file for this particular test.

## 5.3 Detection of Violation Types

Even though we already limited ourselves to spatial memory errors, still more distinctions can be made: like where in memory the violation occurs (e.g., stack, heap, global) and as a result, not all defense systems can catch every type of violation. For this reason it is possible to specify the



type of violation when defining a CVE; likewise, for a defense system, the detectable violation types can also be defined. This way if a defense system fails to detect a violation that it could not have detected, the result will also be marked as not detectable in the generated report, the result table will still say ‘not detected’ however, the background will be green instead of red.

One of the essential features of the framework is to the ability to detect if a defense system can identify a vulnerability in a program that is known to be vulnerable. We use the following approach to solve this problem. The framework contains a set of defense systems and a set of CVEs, for each item in these sets a definition file is present that describes how the framework can use the defense systems and the CVEs. Using these definition files the framework can compile a CVE (or at least the associated program) using a designated defense system. The result is an instrumented binary that should print an error message once the framework triggers the vulnerability in the program. The output of the program can then be used by the framework to determine if the defense system can detect the vulnerability.

The reason we choose this detection method is because of its simplicity, as it does not require modifying any of the source code in both the program from the CVEs or the defense systems. Also, it does not need a complicated way of triggering the violation in the program, as long as, the overflow is triggered it is it can be detected. Other options to determine defense systems performance are also possible. We will quickly discuss these and explain that our implementation is the most flexible and robust.

The first option is to create an exploit for each CVE that will print a characteristic text once it is inserted into the running application. This option, however, has clear disadvantages as it requires a vulnerability that is susceptible to code injection or control flow modification. It is clear that using this method will limit the useable CVEs even more, and also creating exploits for a CVE is more difficult. Furthermore, these kinds of exploits do not have to be system independent or have a 100% chance of succeeding. Because of these reasons, it is clear why we choose to not to use this option.

The second option that can be used to measure the performance of the defense systems is again based on looking at the intended output of the application, rather than the output generated by the instrumentation. However, this time we add a line of code to the program that will print a unique piece of text, that can be used by the framework to determine if the defense system detects the violation. The user should insert the line of code just after the place where the violation happens. The idea is that this the line of code is never executed since the defense system should prevent further execution of the application after a (memory) violation occurs.

The first alternative has too many drawbacks to be considered as a viable method. However, the second option is a solution that has one significant benefit over the method that the framework currently uses. Specifically, the framework knows when the violation has occurred and can use this information to kill the process, in the case that the defense system fails to do this. In some cases, the program can keep running indefinitely, and therefore stalls the framework. The text printed by the program is a natural point to terminate the application that this alternative option can use whereas the current implementation does not know whether the violation has happened and therefore does not know whether it can kill the process. For this reason, we did not implement any features to kill running processes in the framework. The lack of this feature is not much of a problem as it turns out that this problem does not arise very often. If an application is likely to run indefinitely, it is up to the user implementing the CVE to come up with a solution (e.g., built in a timer to kill the process after a while).

However, there is also a substantial drawback to the second alternative discussed above. Namely, the user needs to modify the code of the program. Not only is this inconvenient, it may also influence the results of the tests. For this reason, we choose to use the output of the defense system rather than the output generated by the application itself.

## 5.4 Adding/Changing content

When adding new CVEs to the framework, the users should take some characteristics of the CVE into account. The most important property is that it should be a spatial memory violation that the framework can reliably trigger. Moreover, the project needs to be open source, such that a defense system can compile the source into an instrumented binary. Furthermore, to improve compatibility with other systems and to reduce build time is good practice to choose CVEs on small projects that are well behaved. Meaning that the project should be known to compile easily on multiple operating systems (this is often the case for popular software), and the project should not generate an abundance of compiler warnings. Projects that use an exotic build system are also better left out, as these prove to be problematic to use together with a defense system. Not to mention that this would also require even more additional packages to be installed on the system, making the use of the framework more complex.

It should be easy to add/change content to the framework, as already discussed above. Newly added content is automatically detected and rebuilt without the need to rerun all of the other tests; this feature saves quite some time when someone changes the framework. The way that either a CVE or a defense system is added should be easy, nicely structured and have similarities where possible (i.e., function names, the same member variables names, the structure of the file, etcetera).

We choose the following approach for this design problem. Both the CVEs and the defense systems have a subfolder in the framework root, ‘cve’ and ‘defense’ respectively. In these folders, a new folder should be created for each new CVE/ defense system. For a defense system the name of the folder should clearly indicate the name of the defense (and optionally a version if more versions of the same defense system exist), for example for ‘ASan’ a good choice would be ‘asan\_gcc’, ‘asan\_llvm’ or ‘asan\_llvm\_git’. For the CVE there is a more strict naming convention this should follow: ‘CVE-XXXX-XXXX’, this is because the database itself uses the same convention. Also, note that there are no real restrictions to the filename. However, it is wise to stick with the following character set: ([upper] | [lower] | [digits] | - | \_), and especially spaces should be avoided.

Each of these folders needs to contain at least a `def.py` file, in this python file a class needs to be included that defines the CVE or the defense system. These classes should extend from a base class provided by the framework for either a CVE or a defense system. This base class abstracts the repetitive tasks for the user and also enforces a uniform look between different CVE and defense system definitions. Each of these classes gets passed a `Shell` object that can be used as a shell to run commands, change directory, and so forth. The added benefit of using a specially defined `Shell` class is that the framework has full control over the output of commands it runs, the working directory, etcetera. It should be noted though that commands are not jailed and therefore great care should be taken when writing these CVE/defense definition files.

The question then remains what needs to be defined in these files, for a CVE this boils down to a: name, version and violation type as member variables, along with functions for cleaning, building and exploiting. In these functions, the user can use the shell member variable to call the appropriate commands. The exploit function should also return the output of the command under test.

For the defense systems about the same applies as for the CVEs, again the name and version should be set. If the version number is prone to change or is unknown this variable is best left untouched. In this case, the framework will try to detect a version on runtime. Furthermore, the following other member variables need to be defined: build parameters; this variable holds information where the defense system is installed and how it can be used. Moreover, a member that holds characteristic error messages that are to be used when checking if the defense system

catches a violation. Finally, a list of detectable violation types should also be defined (alternatively this can be set to a special keyword to accept every violation). Optionally functions can be defined that allow the framework to download and build the defense system. These functions work similar to the ones in a CVE definition file.

## 5.5 Data Representation

There should also be a clear way to review the data generated by running the tests. Therefore, the framework has a built-in HTML report generator that will take the result files and generate a report. The generator first creates all of the HTML elements, and then insert these into an HTML template file. Among those elements are two tables and some statistical values like the number of CVEs, the number of defense systems, creation date, among others. The first table contains all of the test results. The second table contains detection percentages per defense system. If the generator finishes, a user can view the generated report in a browser. Alternatively, it is also possible to process the result files directly. For this reason, the format of these files is ‘JSON’ and thus these files are relatively simple to work with using a different application.

# Chapter 6

## Implementation

Figure 6.1 shows a directory listing of the framework, containing four directories at the top level. The first dictionary, **cve** contains all of the CVEs that have been added to the framework, note that the **cve** folder itself also contains directories carrying the name of the CVE. For clarity, the content of these is only shown for **CVE-2017-8364**. The CVE folder is also used for storing files during the test phase to build and run tests for that particular CVE, and these temporary files can be cleaned with the command: **clean**. The next directory in the list is **defense** this folder contains all of the defense systems added to the framework. Similarly to **cve** only the content for one folder, **asan\_gcc** is shown. The third directory is **results** this directory is used to store result files and log files, this folder can be cleaned with the **clean.results** subcommand. The last folder is **framework** this contains the actual python source code for the framework.

Moreover, there are three files: **prepare\_ubuntu18041ts.sh** can be used to install all of the dependencies and make some minor modifications to the OS that the framework requires. In addition **report.htm** is the report that can be generated from the result files. Finally **run.py** is just a small Python script for that is to be used to interact with the framework.

```

framework/
├── cve/
│   ├── CVE-2014-0160/
│   ├── CVE-2017-8361/
│   ├── CVE-2017-8364/
│   │   ├── rzip-2.1.tar.gz
│   │   ├── def.py
│   │   └── poc.rz
│   ├── CVE-2017-9048/
│   ├── CVE-2017-9928/
│   ├── CVE-2017-9929/
│   ├── CVE-2018-16375/
│   ├── CVE-2018-7487/
│   ├── CVE-2018-7648/
│   └── CVE-2018-8905/
├── defense/
│   ├── asan_gcc/
│   │   └── def.py
│   ├── asan_llvm/
│   ├── asan_llvm.git/
│   ├── lowfat/
│   ├── safecode/
│   └── softbound/
├── framework/
│   ├── application.py
│   ├── cve_def_base.py
│   ├── defense_def_base.py
│   ├── non_zero_return_exception.py
│   ├── report_generator.py
│   ├── report_template.htm
│   ├── shell.py
│   └── violation_type.py
├── results/
├── prepare_ubuntu1804lts.sh
├── report.htm
└── run.py

```

Figure 6.1: Shortened directory listing of the framework. The content of the CVE folders, as well as, the content of the defense system folders is for clarity mostly left empty, we only give one example for each of these folders.

## 6.1 Source Files

From the source files in `framework`, `application.py` is the most important, as this is responsible for interpreting the user input. According to this it decides what needs to happen and loads the appropriate CVEs and defense systems. The subcommands `clean_results` and `gen_report` do not require a single CVE or defense to be loaded, as these will act completely based on the content of the results folder. The commands `clean`, `build_defense` and `clean_defense` only require the CVEs or defenses to be loaded respectively. `run_tests` is the only command that

will load a CVE and a defense at the same time, as the `_build` target of a CVE definition requires a `build_param` dictionary that is provided by a defense system. Another task of the `application.py` file is to manage the content of the `results` folder. For example, when running the `run_tests` subcommand it is the responsibility of this source file to store the result and log files.

The `cve_def_base.py` and `defense_def_base.py` contain base classes that need to be extended when either a CVE or defense system is added. These classes implement the tasks that are similar for all cases, thus reducing the need to rewrite code. The file `violation_type.py` is also to be used when defining a CVE/defense system, as this contains an enumeration of the violation types.

`shell.py` contains a class that implements a shell-like interface that is to be used by the CVEs and defense systems. It provides all of the necessary features that are expected to be there i.e. `call` to run a command, `cd` to change directory and `set_env/get_env` to change the environment. These functions wrap the calls to the operating system and also redirect the IO to the correct log file. Furthermore, `non_zero_return_exception.py` contains an exception class that is raised when a shell command returns a non zero value.

The last files in the `framework` directory are `report_generator.py` and `report_template.htm`. The first contains the code to generate the report file. This piece of code will search the content of a result folder and load all of the files with the JSON extension. It will then calculate the necessary values and insert them in the report. The latter file is an HTML report template, the fields that are to be inserted by the report generator have the following format: `{FIELD_NAME}`.

## 6.2 Adding a Defense System to the Framework

Adding defense systems is an easy task and is done by creating a new folder that carries the name of the defense in the `defense` folder of the framework. This folder should contain at least a `def.py` which defines the defense system, this is done by defining a couple of variables in the `Def` class that derives `DefenseDefBase`. The `_name` variable should be set to a string that contains the name of the defense system. Optionally it is possible to define a `_version` variable string. In the case that this variable is not defined the framework will attempt to determine the version on runtime, this is particularly useful when the version is unknown or prone to change. Perhaps the most important variable is `_build_params` this tells the framework where the compiler collection is located at the device and which compiler/linker flags to use. All of these properties are saved in a python dictionary.

Another important variable is `_error_messages` this contains a list of keywords that will be checked against the output of a program to determine whether a defense system was able to detect the violation. Since the output of a program is used to determine if the defense succeeded, care should be taken to prevent false positives when adding a CVE to the framework that outputs similar data. A simple solution can be to suppress the standard output stream.

Similar to the previous variable a list should be defined to tell the framework which violation types it can detect. This variable has the name `_detectable_violations`, it is possible to assign the value `ViolationType.ALL` to match all violation types.

Optionally it is also possible to define the following two functions: `_clean` and `_build`. These functions can use the `_shell` member variable to define a series of commands on how to clean and build the defense system. Note that for defense systems that are installed (globally) on the operating system these functions are not needed, instead the `_build_params` should contain the appropriate static paths. For clarity, the Asan/llvm `def.py` is included below:

```

from framework.violation_type import ViolationType
from framework.defense_def_base import DefenseDefBase

class Def(DefenseDefBase):
    def __init__(self, shell):
        super().__init__(shell)

        self._name = 'ASan_(llvm)'
        self._version = '7.0.1'
        install_dir = '{}/{}/{}'.format(self._shell.get_cwd(),
                                         '/install/bin')
        clang_path = '{}/{}/{}'.format(install_dir, 'clang')
        clangpp_path = '{}/{}/{}'.format(install_dir, 'clang++')
        self._build_params = {
            'cc': clang_path,
            'cxx': clangpp_path,
            'cflags': '-fsanitize=address',
            'ldflags': '',
        }
        self._error_messages = [
            'stack-buffer-overflow',
            'global-buffer-overflow',
            'heap-buffer-overflow']
        self._detectable_violations = ViolationType.ALL

    # some commands are omitted
    def _build(self):
        super()._build()

        root_dir = self._shell.get_cwd()
        src_dir = '{}/src'.format(root_dir)
        build_dir = '{}/build'.format(root_dir)
        install_dir = '{}/install'.format(root_dir)

        self._shell.call('mkdir_src_build_install')
        self._shell.cd(src_dir)
        self._shell.call('curl_sOL...')
        self._shell.call('tar_xvf...')
        self._shell.cd(build_dir)
        self._shell.call('cmake_G"Unix_Makefiles"...')
        self._shell.call('make_j4')
        self._shell.call('make_install')
        self._shell.cd(root_dir)
        self._shell.call('rm_rfv_src_build')

    def _clean(self):
        super()._clean()
        self._shell.call('rm_vrf_src_build_install')

```

### 6.2.1 Address Sanitizer

The implementation of GCC address sanitizer was the most straightforward. Because it is already installed on most machines, there was no need to compile anything. Instead, the existing compiler can just be added to the framework. ASan only requires a single flag (`-fsanitize=address`) to be specified to enable the defense.

ASan for LLVM (7.0.0 and the git version) are also implemented, however LLVM is compiled for from source locally with ASan enabled. The `def.py` file looks very similar to the one for GCC, except that the path to the compiler is now a dynamically generated directory and the build functions are populated.

### 6.2.2 LowFat

Getting LowFat to compile required some changes to the source code, at least on Arch Linux the OS where the framework was initially developed on. However later when we moved to Ubuntu, LowFat compiled without any problems. Therefore the issues discussed only apply to the Arch Linux system.

The code comes directly from GitHub [6]. The first adjustment is that `struct udev` is no longer available; however, the size of this structure is still used in some parts of the source. An ad hoc fix is to define the structure locally; this did not pose any problems since this was only necessary during development. On the reference Ubuntu machine this problem did not arise. Another compiler error is generated by a wrong cast from unsigned char to char, in the ‘OrcRemoteTargetClient.h’ file. Changing char to unsigned char on line 690 solves the problem. After these adjustments the source compiles and the compiler can be added to the framework.

The `def.py` looks almost the same as ASan/LLVM, the only difference being that the compiler flag becomes `-fsanitize=lowfat`, the path to `clang` changes to the LowFat install dir. LowFat also requires linking against its library; thus some linker flags to the LowFat library are necessary.

Furthermore, LowFat requires BMI (bit manipulation instructions) to be available on the machine it is running on if it does not detect this a legacy build will be created. This mode is not supported anymore and may influence the final results slightly. Unfortunately, the machine used for testing does not support these BMI instructions. Therefore, all results are for LowFat are based on the legacy version.

### 6.2.3 SafeCode

SafeCode had many problems when compiling on a recent system. However, most of them were minor and could easily be solved. For example, the project is compiled with `-Werror` this causes the compiler to handle a warning as an error, disabling this flag already solves some problems. There were also many small fixes required to get the compiler to compile. These fixes can be found in the `safecode_30_ubuntu_1804.patch` (see Appendix B). Adding this defense to the framework follows the same routine as the previously discussed defense systems, the compiler flag to enable SafeCode is `-fmemsafety`. It is also necessary to provide a linker flag that specifies the location of the SafeCode (runtime) libraries. The source code for SafeCode can be found here [5].

### 6.2.4 Softbound

Softbound can be downloaded here [15], the project compiled without any issue on Arch Linux and installing it in the framework is once again similar to those mentioned above. The only difference being the compiler flag: `-fsoftboundcets` and the different linker paths. On Ubuntu



there where some problems while compiling, a patch file is created to solve these and can be found under the name: `softbound_30_ubuntu_1804.patch` (see Appendix C).

### 6.2.5 Failed Defense Systems

We have also tried compiling Deltapointers [13] and this eventually functioned albeit in a virtual machine with Ubuntu. The build system did not work well with Arch Linux, requiring us to build the project in a virtual machine. Another problem is the lack of BMI instructions on the development computer; this required rewriting a couple of mnemonics in C++. Resulting in a significant performance loss when using the defense system. The performance loss is however not the main reason why we left this defense system out, because runtime overhead is not the main focus of this research.

The main problem is that the defense system requires a different way of compiling an application. It is not just enabling an additional compiler flag and running it. To our knowledge the C/C++ code first needs to be compiled into LLVM IR, subsequently this needs to be modified by some LLVM passes and then be compiled into a binary. This process would require more work to add to the framework as a tool is needed to perform the steps mentioned above while presenting a Clang/GCC interface.

## 6.3 CVE definition

Adding a CVE to the framework is similar to adding a defense system, it also requires creating a new folder containing a `def.py` file. However, the folder should be created in the `cve` folder. In the `def.py` the following functions should be overloaded. Firstly, `_clean` function that removes all files resulting from building and running the exploit. Secondly, `_build` function that builds the source using the selected defense system. The compiler to be used is found in the `build_params` dictionary using the keys `cc` and `cxx` for the C and C++ respectively. The required compiler and linker flags can also be found in the dictionary mentioned above using the keys `cflags` and `ldflags`. Finally, a `_exploit` function should be defined that runs the program and triggers the exploit. This function should return the output of the application making it is possible to check if the violation is detected. Along with these functions the variables `_name`, `_version`, and `_violation_type` should also be defined containing the name of the CVE, the version and the violation type respectively. All of these function/variables are members of the `CVEDefBase` class that should be extended by a class called `Def`. This base class stores the `_shell` object that is to be used to call the commands.

When a program is called from one of the three functions, and it fails, (i.e., returns a non zero value) the `call` function throws a `NonZeroReturnException` exception and this terminates the testing process prematurely. It is possible to allow programs to fail while still carrying on the test process. However, this needs to be specified explicitly using the optional `allow_failure` parameter of `call`. In fact, most defense systems return a non zero exit code when they detect a violation; therefore, it is almost always necessary to specify this parameter when running the exploit command.

If a CVE is successfully added it may be wise to check the log file for the not detected cases and verify that no false negative is generated. Likewise failed to build should also be checked to see if the build did not fail for some configuration error, e.g., a wrongly specified compiler flag.

### 6.3.1 CVE-2014-0160

CVE-2014-0160 is the well known Heartbleed exploit in OpenSSL (version 1.0.1g), this is by far the most exciting CVE in the collection. This has various reasons one being that it is so well known and has become a norm to include this exploit in papers about OOB defense systems. Another interesting fact about this bug is that a server-client setup is needed to trigger the exploit. This implementation serves as an example of how to implement such a combination in the framework.

The vulnerability can be found in the files `t1_lib.c` and `d1_both.c` however, we will focus on the first file only. In this file, the heartbeat extension [20] is implemented among others. This extension can be used to keep a connection alive when there is no continuous data transfer. The basic idea is to send, after some time, a small request packet (a heartbeat). Subsequently, the receiver will send a response back, containing the same payload as the request (i.e., a copy). Certainly, there are some more formalities involved like how to handle a malformed packet, timed-out packet, and so forth; however, this knowledge is not necessary to understand the vulnerability. As one might expect the error lies within the copying of the request payload to the response payload, that happens when a response message is built to answer a request. The value of the `payload_length` is never checked and thus the consecutive `memcpy` command will happily copy everything after the original payload to the response message payload field. The data type of the `payload_length` is a unsigned integer of 2 bytes so the maximum amount of bytes that can be leaked per 65535 bytes <sup>1</sup>. This entire spectacle can be found in the `tls1_process_heartbeat` function in the aforementioned C file.

In order to exploit this bug one needs to send a crafted heartbeat packet to a vulnerable server. This requires some unconventional methods as sending arbitrary data to a connected SSL server is (for good reasons) not supported by OpenSSL. Fortunately, OpenSSL is written in C and thus calling `ssl->method->ssl_write_bytes(ssl, TLS1_RT_HEARTBEAT, buffer, buffer_len)` directly is possible. Here `ssl` is a connected SSL object to the server, the second argument specifies the type of message, and finally `buffer` contains the crafted heartbeat request packet. The next thing to do is to wait for incoming data; this can easily be achieved by calling `SSL_read` continuously on the `ssl` object. However, this does not return the heartbeat response data as this is not part of the actual data that is communicated between client and server. A quick way around this is to once again delve into the `ssl` object and override the `msg_callback`. The code section below contains relevant parts of the client side exploit program included with the CVE. Note that the `alarm` is only to terminate the application after a couple of seconds. The full source file can be found under the name of `exploit.c` in the CVE folder of this vulnerability.

```
static size_t heartbleed_pkt(...)
{
    unsigned char *p = (unsigned char *)buf;

    *p++ = 0x01; // HB Request
    *(unsigned short *)p = htons(hb_len);
    p += 2;

    // padding count to 16
    for (unsigned char i = 0; i < 16; i++)
    {
        *p++ = i;
    }
}
```

---

<sup>1</sup>or  $2^{16} - 1 - \text{min\_padding\_length} = 2^{16} - 1 - 16 = 65519$  bytes to be pedantic.

```

    return 3 + 16;
}

int main()
{
    ...

    char buffer[128];
    size_t buffer_len;

    // create crafted heartbeat message
    buffer_len = heartbleed_pkt(buffer,
        sizeof(buffer), OVERFLOW_SIZE);

    ssl->msg_callback = hex_dmp_callback;
    ssl->method->ssl_write_bytes(ssl,
        TLS1_RT_HEARTBEAT, buffer, buffer_len);
    alarm(5);
    for (;;)
    {
        SSL_read(ssl, buffer, sizeof(buffer));
    }

    ...
}

```

Adding this CVE to the framework did also pose a small challenge since this requires a server and a client to run and to be shut down correctly, which is not supported by the framework (only one command can be executed at a time). However, it is possible to create a small shell script that handles the starting of the server and then the client. In this script, the output of the client program is silenced because this does not add any value and may even disrupt the defense system output generated by the server. To make sure that the server is shut down ‘correctly’ a `pkill -9 $(jobs -p)` is also run at the end of the script.

As can be seen in the listing, the exploit program uses some internal constants and functions of the OpenSSL library, and as a result, it will break when trying to compile against a more recent version of OpenSSL. To solve this problem, the OpenSSL library gets compiled twice when this CVE gets tested by the framework. The server gets compiled with the currently selected defense system and the client gets compiled with the default compiler.

### 6.3.2 CVE-2017-8361

CVE-2017-8361 is a global buffer overwrite in libsndfile version 1.0.28, the vulnerability is triggered via a crafted audio file. Adding this CVE to the framework was easy as there was already a POC file provided here[19]. Also, because setting an environment variable is enough to enable a custom compiler, building this project with a defense system is trivial. If the compiler completes triggering the vulnerability can be achieved by running:

```
$ ./sndfile-convert $POC out.wav
```

### 6.3.3 CVE-2017-8364

CVE-2017-8364 is a heap buffer overwrite in rzip version 2.1, resulting from an inadequate check when reading the header of a rzip file. Therefore, the vulnerability can be triggered by loading a crafted file.

We have created such a file using the following method. The CVE description tells us that the `read_buf` function, where the violation occurs, is in `stream.c` file. Looking at the function prototype we see that it takes: a file descriptor, a buffer to write, and a size. By searching through `stream.c` for calls to `read_buf` it is possible to trace where the size gets read from the header of the rzip file that is being loaded. This happens to be in the `fill_buffer` function located in the same file. Printing the cursor position of an open file reveals the location in the rzip file header where the buffer size is being saved. Changing this value to a value larger than the buffer in the file (i.e., the value that it is currently at) will result in a file that can be used to trigger the vulnerability.

Alternatively, it would also have been possible to look up the header structure and thus the position of this value. Regardless, the method currently used does not require any knowledge about the used file structure. To trigger the vulnerability the following command can be used, where `$POC` is the just created crafted file.

```
$ ./rzip -fkd $POC
```

### 6.3.4 CVE-2017-9048

CVE-2017-9048 is a stack buffer overwrite in libxml2 version v2.9.4-rc2. Adding this CVE to the framework was easy since there was already a POC provided here[3]. Compiling with the different defense systems did not pose any significant problems. The following command can be used to trigger the vulnerability:

```
$ ./xmllint --valid $POC
```

### 6.3.5 CVE-2017-9929

CVE-2017-9929 is a stack buffer overwrite in lrzip version 0.631, once again adding this vulnerability was easy since a POC was already available and can be found here [12]. Compiling and running did also not pose any major problems.

### 6.3.6 CVE-2017-9928

This CVE has many similarities with CVE-2017-9929 including application and version number. A POC regarding this CVE can be found here [11].

### 6.3.7 CVE-2018-16375

CVE-2018-16375 is a heap buffer overwrite in OpenJPEG version 2.3.0 triggered via a crafted PPM image file. PPM is a straightforward image file format that may use ASCII characters to describe the image. The vulnerability can happen because of insufficient checks while reading the file header. This can lead to an integer overflow while allocating the buffer due to multiplication overflow. When parsing the data, this integer-overflow does not occur and therefore a buffer overwrite can happen. The following (pseudo) code sketches a simplified version of the bug, the real code can be found in the function `pnmtoimage` in `bin/jpwl/convert.c`.

```

void read_pnm_header (...)
{
    int format, width, height;
    // these calls are checked for overflows, invalid input etc.
    format = read_format(fd);
    width = read_int(fd);
    height = read_int(fd);

    ...

    // integer overflow possible
    char *buffer = (char *)malloc(width * height);

    ...

    if (format == 4) {
        // no overflow because of double for loop
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                //violation happens here
                buffer[x + y * width];
            }
        }
    }
    else if (...) {} // rest of the cases are omitted
}

```

It can be seen that it is possible to choose *width* and *height* such that the call to `malloc` gets passed a reasonably small value while the  $x + y * width$  can take on much larger values thus resulting in a heap buffer overflow.

Creating a file that triggers the vulnerability is almost trivial, all that needs to be done is to create a PMM file (either from scratch or with some image manipulation program) and then modify the following. Change the format to 'P4', this is important since other formats use a single for-loop that loops until  $width * height$  and thus the same integer-overflow prevents the vulnerability from triggering. The last step would be to change the 'width' and 'height' to something that will overflow the `int` datatype.

By default the JPWL extension of the library is disabled, this can be enabled by specifying the following cmake flag: `-DBUILD_JPWL=ON`. The following command can be used to trigger the vulnerability:

```
$ ./opj_jpwl_compress -i $POC -o image.j2k ; rm -f image.j2k
```

### 6.3.8 CVE-2018-7487

CVE-2018-7487 is a heap buffer overwrite in `sam2p` version 0.49.4, this is an image converter program. The violation happens in `loadPCX` function in `in_pcx.cpp`. The violation can be triggered using a crafted image file that can be found here [23]. Adding this CVE to the framework required some trial-and-error, due to the different behavior of the compilers in which a configuration that works on one defense systems breaks on the others and vice versa. Also, because this project does not use an automated build system changing the linker/compiler flags has to be

done using a `sed` command to modify the Makefile directly. The exploit can be triggered using the following command:

```
$ ./sam2p $POC EPS: /dev/null
```

### 6.3.9 CVE-2018-7648

CVE-2018-7648 is a stack buffer overwrite that happens in the main function of the following file `opj_mj2_extract.c`. This is a small helper program in the OpenJPEG project version 2.3.0. Triggering the vulnerability is trivial, as the classic `sprintf` example is implemented here in real-life when parsing the command line option that specifies the output filename. In the code, the buffer that stores the filename has a size of 50 characters so naturally when entering a filename that is greater than this the violation triggers, for completeness the command to achieve this is stated below:

```
$ ./opj_mj2_extract $INPUT $OUTPUT
```

Where the length of `$OUTPUT` needs to be greater than 50, also note that the input needs to be a valid ‘OPJ’ movie. The one provided in the framework was created using a short clip of the “Big Buck Bunny” movie.

### 6.3.10 CVE-2018-8905

CVE-2018-8905 is a heap buffer overwrite, in LibTIFF version 4.0.9. The violation happens in `tif_lzw.c` in the function `LZWDecodeCompat` and can be triggered using a crafted ‘TIFF’ file. A POC was already available and can be found here: [10]. Getting this CVE to work with the framework was straightforward, the following command can be used to generate the violation.

```
$ ./tiff2ps $POC
```

## Evaluation

Using the described framework it becomes clear that it is well possible to test defense systems against real-world vulnerabilities, in the form of CVEs. Moreover, even a great deal of the process can be automated after the content has been added. An image of a (partial) report can be seen in Figure 7.1, from this it is clear that the results are surprising. The overall result for a specific defense system varies from detecting almost everything to not being able to even compile half of the CVEs out of the test set. Therefore, we discuss each defense system separately evaluation below.

It should also be noted that the framework was developed at a different Linux distribution (Arch Linux) than the one used for generating the final results. Nevertheless, the results are very similar. Slight variances that do emerge are, for example, on the Arch system ASan would score an astonishing 100% detection rate, where on Ubuntu it misses one CVE. Surprisingly, for this particular CVE, the roles are reversed concerning Softbound, and it does get detected on Ubuntu where it did not work on Arch Linux. Another thing to note is that Ubuntu has more library incompatibilities than Arch Linux. However, we still choose to include the Ubuntu results because these are much easier to reproduce. Especially Softbound and to a lesser extent, Safecode suffer from these library incompatibilities.

### 7.1 Address Sanitizer

It can be seen that ASan, with a 92% detection rate, has the highest detection rate of all evaluated defense systems. This high rate has two primary reasons, for one ASan is more of a sanitizer than a defense system and therefore focuses on accurate detection rather than having low runtime overhead. This difference is something the framework is not able to distinguish very well, as there are no built-in timing benchmarks. The other reason is that ASan has become the industry standard when it comes to detecting (memory) violations. Resulting in an enormous bias towards ASan detectable violations in the CVE database, as almost all of the memory related CVE are provided with a POC that works with ASan.

In the final results, there are three different compilers that all use ASan there is no difference between any of these versions. Nevertheless, this is still an interesting result. The ‘git’ version was added later when ASan failed to detect **CVE-2017-8361**, one possibility was that the version used on Arch Linux was newer. To verify this hypothesis we added LLVM-git to the framework unfortunately, this did not appear to solve the missing CVE problem. We assume it is a

## Detection results

	asan_gcc ASan (gcc) (7.3.0)	asan_llvm ASan (llvm) (7.0.1)	asan_llvm_git ASan (llvm git) (9.0.0)	lowfat LowFat (4.0.0)	safeocode Safeocode (3.0)	softbound softbound (3.4)
<b>00_heap_oob</b>						
heap_oob (Example) (0.0.1)	detected (0.2)	detected (0.4)	detected (0.5)	detected (0.4)	detected (0.5)	detected (0.6)
<b>00_stack_oob</b>						
stack_oob (Example) (0.0.1)	detected (1.6)	detected (0.5)	detected (0.5)	detected (0.4)	detected (0.5)	detected (0.8)
<b>CVE-2014-0160</b>						
OpenSSL (1.0.1f)	detected (347.4)	detected (357.8)	detected (367.2)	detected (406.3)	not detected (306.0)	build failed (580.4)
<b>CVE-2017-8361</b>						
libsndfile (1.0.28)	not detected (67.5)	not detected (88.5)	not detected (90.1)	not detected (88.2)	not detected (56.3)	detected (179.4)
<b>CVE-2017-8364</b>						
rzip (2.1)	detected (3.6)	detected (8.2)	detected (8.9)	not detected (6.6)	detected (3.8)	not detected (7.9)
<b>CVE-2017-9048</b>						
libxml2 (libxml2-v2.9.4-rc2)	detected (122.8)	detected (136.9)	detected (138.2)	not detected (142.9)	detected (97.3)	build failed (236.9)
<b>CVE-2017-9928</b>						
lrzip (0.631)	detected (52.1)	detected (72.3)	detected (70.6)	not detected (72.9)	detected (48.0)	build failed (96.1)
<b>CVE-2017-9929</b>						
lrzip (0.631)	detected (52.3)	detected (75.8)	detected (74.5)	not detected (75.1)	detected (49.5)	build failed (103.6)
<b>CVE-2018-16375</b>						
OpenJPEG (2.3.0)	detected (45.2)	detected (51.1)	detected (52.9)	detected (58.8)	detected (32.0)	build failed (112.1)
<b>CVE-2018-7487</b>						
sam2p (0.49.4)	detected (34.8)	detected (35.2)	detected (35.2)	detected (25.7)	build failed (0.7)	build failed (16.7)
<b>CVE-2018-7648</b>						
OpenJPEG (2.3.0)	detected (44.5)	detected (53.4)	detected (54.3)	not detected (57.0)	detected (31.6)	build failed (111.8)
<b>CVE-2018-8905</b>						
LibTIFF (4.0.9)	detected (60.8)	detected (81.1)	detected (83.8)	detected (80.1)	build failed (37.0)	detected (165.8)

## Total detection percentage per defense

	Detected	Expected	Not detected	Build failed
<b>asan_gcc</b>	92%	92%	8%	0%
<b>asan_llvm</b>	92%	92%	8%	0%
<b>asan_llvm_git</b>	92%	92%	8%	0%
<b>lowfat</b>	50%	58%	50%	0%
<b>safeocode</b>	67%	67%	17%	17%
<b>softbound</b>	33%	33%	8%	58%

Figure 7.1: This figure shows the results table taken from an automatically generated report. These tests were carried out on a clean Ubuntu 18.04 LTS machine with only the base system installed, except that the `prepare_ubuntu1804.sh` script that ran once. The numbers between parenthesis show the compile and runtime of a test.

dependency problem, but this remains unverified.

## 7.2 LowFat

Even though LowFat only has a 50% detection rate, it is still a competent defense system; this has to do with the fact that initially LowFat was developed for heap memory errors and only at a later time stack (and partial global) memory space where added [9]. Unsurprisingly, 4 out of 5 ‘not detected’ CVEs are stack buffer related violations (only CVE-2017-8364 is a heap buffer overwrite). Since the stack memory error detection functionality is enabled stack memory errors are marked with a red background. To verify that the functionality is working all someone needs to do is look at the result of `00_stack_oob` gets detected, for LowFat this does, in fact, get detected.

Another thing to note about the results for LowFat is that it has been built using a CPU that does not support BMI instructions. As a consequence, it gets built as a legacy version. This version is no longer supported, but mostly has the disadvantage that the binaries generated by the defense system run slower.

In short, when looking for a defense system that has excellent heap protection, LowFat is



an excellent choice. Primarily because it acts as a drop-in replacement compiler without much difficulty (i.e., almost no problems arose during compilation of the CVEs).

### 7.3 SafeCode

SafeCode is a defense system with quite a high detection rate. However, there are some minor problems with compiling some of the CVEs. As this is a somewhat older defense system, such issues are prone to arise on a newer OS. In fact, both of the failed tests are due to incompatibilities with the operating system. Solving them is probably possible by changing with compiler flags; modifying the source code slightly; changing the build system of the source code one want to compile. Except for these minor inconveniences, it should also be taken into consideration that without patching, SafeCode does not compile on Ubuntu anymore. Because of the high detection rate of SafeCode, it may be worth considering it, even though it has some compatibility problems.

### 7.4 Softbound

Finally, there is Softbound this is by far the worst performing defense system in the bunch. A real conclusion on detection rate is hard to draw since most of the tests can not come past the compilation stage. About half of the tests fail to compile due to a crash of clang, the other half is because of incompatibilities with Ubuntu<sup>1</sup>. The reason is, just as with Safecode, that this is an older defense system that has not been maintained for the past seven years. So unless one has a good reason to use Softbound it should be avoided.

---

<sup>1</sup>On Arch Linux this was a little better, but not much, only one more test managed to build successfully.

## Discussion

The question of whether we succeeded in creating a generic framework that is capable of supporting multiple defense systems and multiple CVEs is easily answered. Yes, this can be done using our newly developed framework described in this thesis. In order to support multiple defense systems and multiple CVEs, the interface of a certain compiler (when adding a defense system) or the interface of a build system (when adding a CVE) is converted to meet the framework specification. This conversion happens in the `def.py` files provided with the framework. The specification of the framework is chosen such that it is generic enough to fit almost all defense systems and build systems; this choice results in short and clear conversion scripts (i.e., `def.py` files) scripts that can easily be understood.

Using the provided defense systems and CVEs our framework can generate a table that can be used to check a defense systems performance, that is how many vulnerabilities it can detect. These performance numbers can then easily be compared to the values obtained by the other defense systems in the framework. Looking at the results, ASan is by far the best performing defense system, followed by Safecode, LowFat and finally Softbound respectively.

Moreover, by looking at the ‘build failed’ cells it is possible to check the compatibility (by this we mean the ability to act as a drop-in replacement for a regular compiler) of the defense system. By looking at the log files, it is also possible to determine whether the error originates from library/dependencies incompatibilities or because the defense system generates an internal error (e.g., a crash). The result for compatibility are as follows: ASan and LowFat are the most compatible defense systems and manage to build every CVE successfully, Safecode only fails to build two CVEs, and again Softbound is the worst performer.

### 8.1 Generated Results

The set of CVEs currently included in the framework counts 10 CVEs; this is too small a number of tests to base a hard conclusion on, and therefore we do not make claims such as “Defense system A is better than defense system B”. Another reason is that the detection rate is not the only criteria to base such a conclusion on. However, we agree that the number of CVEs gives a good impression of what a defense system is capable of, and therefore a loose comparison is possible.

Another problem that is not picked up by the framework is the possibility to circumvent a defense system after it is known to be used. The reason is that this is hard to test automatically,

as each defense systems requires a different approach or it may be impossible to evade a defense system. Nevertheless, there is another side to the story; the fact that a defense system misses a violation also suggest that a possible way to circumvent the defense system is found. By studying what is happening is may be possible to transfer the behavior to another violation that does get detected and evade the defense system this way.

In the framework, it is possible to specify a violation type for a defense system and a CVE. Currently, the division is made for stack/heap/global reads and writes. This selection can be made even more specific or broader. We choose these distinctions because these three memory regions require different ways of allocating memory and it often occurs that special care needs to be taken to include all three sections in a single defense system. The read and write distinction does not make much difference for the current set of defense systems; however, for future compatibility, it may be an interesting option to have. For example, a defense system/ hardening technique that would benefit from this is the GCC stack protector<sup>1</sup>.

## 8.2 Technical Shortcomings and Further Work

As of right now, only spatial violations are detected using this framework even though there are many other kinds of other violations (e.g., temporal memory errors). Needless to say, it would be beneficial to have also implemented these in the framework. The reason this is not done stems from the time available for this thesis. Adding a different kind of violation type would require adding more defense systems, and also a completely new set of CVEs. However, if one decides to extend the framework data sets with a different kind of violation type, minimal adjustments to the framework are needed. Adding the new violation type to the `ViolationType` class should be enough. Because of the way the framework is set up, defense systems that do not support this type will generate a ‘not detected’ with a green background.

Another improvement to the framework that would yield impressive results, is the addition of an artificial test set (i.e., manually created memory violations) either in existing software or in newly created toy programs. This addition has the benefit that the user has 100% control over what is happening. The idea is partially taken from [26] and this is a good source for possible violation examples. However, the paper is somewhat outdated and discusses more specific return address attacks. The result that this improvement generates will be particularly useful because, if implemented correctly, it will pinpoint the exact shortcomings of a defense system (e.g., LowFat will fail to detect interior OOB violations, whereas Safecode might notice this).

An additional feature that is useful for the extension discussed above, but also for the tests that are currently in the framework, is the possibility to have multiple `_exploit` functions in a CVE definition file. These functions can then trigger the violation slightly different each time, say 1-byte overflow, 1024-byte overflow, and so on. Another approach is to pass an iterating variable in the `_exploit` function; this variable can then be used to decide how to trigger the violation. Alternatively, a combination of the two suggested implementations can be used. The user is then responsible for choosing the most natural implementation when defining a CVE `def.py` file. This addition may seem like it will slow down the testing process tremendously, or clutter the result file. Though, this is not the case the `_exploit` function is usually much faster then the other steps (`_build`, `_clean`). Furthermore, a good way to represent the new results is to replace ‘detected’/‘not detected’ with ‘50% detected’. Where the percentage denotes the number of exploit functions called that were ‘detected’ divided by the total number of calls to an exploit function.

---

<sup>1</sup>If you are wondering how this would expire, so were the authors, as it turns out not too well: missing every possible stack overwrite in the test set.

# Bibliography

- [1] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.
- [2] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting critical elements of stacks. White Paper, Bell Labs, Lucent Technologies, 1999.
- [3] Marcel Boehme. Cve-2017-9048 poc. <https://www.openwall.com/lists/oss-security/2017/05/15/1>, May 2017.
- [4] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. *Special Interest Group on Programming Languages (SIGPLAN) Not.*, 41(6):144–157, June 2006.
- [5] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode source code. <http://safecode.cs.illinois.edu/downloads.html>, November 2016.
- [6] Gregory J. Duck. Lowfat source code. <https://github.com/GJDuck/LowFat>.
- [7] Gregory J. Duck and Roland H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 2016 International Conference on Compiler Construction (CC)*, 2016.
- [8] Gregory J. Duck and Roland H. C. Yap. An extended low fat allocator api and applications. *Computing Research Repository (CoRR)*, abs/1804.04812, 2018.
- [9] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *Proceedings of the 2017 Network and Distributed Systems Security (NDSS) Symposium*, 2017.
- [10] halfbitteam. Cve-2018-8905 poc. [https://github.com/halfbitteam/POCs/blob/master/libtiff-4.08\\_tiff2ps\\_heap\\_overflow/poc](https://github.com/halfbitteam/POCs/blob/master/libtiff-4.08_tiff2ps_heap_overflow/poc), March 2018.
- [11] Con Kolivas. Cve-2017-9928 poc. <https://github.com/ckolivas/lrzip/issues/74>, May 2017.
- [12] Con Kolivas. Cve-2017-9929 poc. <https://github.com/ckolivas/lrzip/issues/75>, May 2017.

- [13] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the 2018 EuroSys Conference*, 2018.
- [14] MITRE Corporation. About CVE. <https://cve.mitre.org/about/index.html>.
- [15] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound source code. <http://www.cis.upenn.edu/acg/softbound/>, November 2011.
- [16] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *Special Interest Group on Programming Languages (SIGPLAN) Not.*, 44(6):245–258, June 2009.
- [17] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Special Interest Group on Programming Languages (SIGPLAN) Not.*, 42(6):89–100, June 2007.
- [18] Alexander Potapenko. Address sanitizer performance numbers. <https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers>, August 2015.
- [19] Agostino Sarubbo. Cve-2017-8361 poc. [https://github.com/asarubbo/poc/blob/master/00265-libsndfile-globaloverflow-flac\\_buffer\\_copy](https://github.com/asarubbo/poc/blob/master/00265-libsndfile-globaloverflow-flac_buffer_copy), April 2012.
- [20] R. Seggelmann, M. Tuexen, and M. Williams. Transport layer security (tls) and datagram transport layer security (dtls) heartbeat extension. RFC 6520, Internet Engineering Task Force, February 2012.
- [21] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.
- [22] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. *Computing Research Repository (CoRR)*, abs/1806.04355, 2018.
- [23] Péter Szabó. Cve-2018-7487 poc. <https://github.com/pts/sam2p/issues/18>, February 2018.
- [24] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.
- [25] Elgar R. van der Zande. Defense System Testing Framework. <https://github.com/elgarvdzande/Defense-System-Testing-Framework>.
- [26] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 2003 Network and Distributed Systems Security (NDSS) Symposium*, 2003.

# Appendix A

## Using the Framework

### A.1 Building the Defense Systems

Building the defense systems is a step that should only be done once; however, because it is such a tedious process, the framework automates this step on Ubuntu 18.04LTS systems. To start this process, execute the command below, where `$framework_root` is the top level directory of the framework.

```
$ cd $framework_root
$ sudo ./prepare_ubuntu1804lts.sh
$ ./run.py build_defense
```

The `prepare_ubuntu1804lts.sh` script is a shell script that will install all of the required packages to build the defense systems and to compile all of the tests. It will also create two soft links of `crti.o` and `crtt.o` in `/usr/lib` such that Safecode will be able to find them<sup>1</sup>. If this script has completed successfully, there is no need ever to rerun it on the system.

The next command is responsible for actually downloading and compiling the defense systems. This command will first clean any files left from a previous build and then recompile the defense systems. It is possible to remove temporary files after the build process, to free up some space. However, this is not a necessary step and is therefore not implemented for all defense systems.

Finally, if a user is only interested in a subset of the provided compilers or wants to compile a compiler that was added later to the system, it is not convenient to recompile all of the defense systems. For this reason, it is possible to augment the third command with the following option `--defense some_defenses` where `'some_defense'` is either a comma-separated list of defenses or the keyword `'all'`. Note that the list can contain just one defense. Moreover, `'all'` will select all available defenses; this is also the default for this parameter. Using this parameter will limit the build process to the selected defense systems only.

### A.2 Running Tests

To run the tests in the framework the following command can be used:

---

<sup>1</sup>The creation of the symlinks can be disabled by commenting the last two lines. However, this results in Safecode performing worse.

```
$ cd $framework_root
$ ./run.py run_tests
```

This command will run all unfinished tests and store their results in the **results** folder. Incomplete tests are tests that do not have a result in the results folder yet. These unfinished tests can happen because of multiple reasons: it is the first time the framework runs; a CVE or defense has just been added to the framework; **clean\_results** has just been run. In other cases **run\_tests** will not do much good and the use of **--build {missing|all|failed}** option is required to force the framework to either rerun all tests or only rerun failed tests. Failed tests are tests that have a 'build failed' status. **missing** is the default option for **--build** and is assumed when nothing is specified. **failed** can be particularly helpful when changes are made to the framework. For example, when adding a CVE, one of the defense systems can fail to compile the newly added CVE. After applying some fixes, using the **failed** build option, it is possible only to recompile using this particular defense system.

There is, however, a small catch to the use of the **all** and **failed** options, namely not only the newly added tests are rerun, but also all of the other tests that match the criteria, leading to the unnecessary use of computational resources and time. Therefore, it is possible to rebuild only a selected number of tests using the **--cve** and **--defense** parameters. For example, to build the CVE-2014-0160 and CVE-2017-8364 using only the Asan/gcc the following command can be used:

```
$ cd $framework_root
$ ./run.py \
    --cve CVE-2014-0160,CVE-2017-8364 \
    --defense asan_gcc \
    --build all
```

### A.3 Generating a Report

```
$ cd $framework_root
$ ./run.py gen_report
```

The commands above will generate an HTML report which can be found in the framework root directory under the name: **report.htm**. This file contains a color-coded table representing the results of the tests. Each cell contains one of the following texts: 'detected', 'not detected' or 'build failed'. 'Detected' implies that the defense system detects the vulnerability this result will always have a green background. 'Not detected' entails that the defense system was unable to identify the vulnerability, this outcome will either have a red or green background. The red background is used when detection failed when this is not allowed (the defense systems claims to be able to catch such violation), a green background occurs when the defense does not claim to be able to detect such violations. 'Build failed' happens when something goes wrong during the test phase (this can occur in all 3 test steps: **\_clean**, **\_build** and **\_exploit**), this outcome will always have an orange background color.

Furthermore, the report also contains a table of statistics per defense system, like detection rate, expected detection rate, 'build failed' rate, and 'failed to detect' rate. Most of the fields speak for them self; however, 'expected' may not be so obvious. The 'expected' field is, for the most part, the same as the detection rate field; however, tests that are 'not detected' and are also not expected to be detected (i.e., the defense system does not support this kind of violation) are also included in this statistic.

## A.4 Cleaning up

There are a couple more sub-commands that can be used to clean the framework. Firstly, `clean` can be used to clean up the build files. Secondly, `clean_results` can be used to clean the result files (including log files), but excluding the report. However, updating the report will require all tests to be rerun. Finally, to remove all of the defense systems `clean_defense` can be used. The clean flags can be useful to save space or reset the framework; another application may be when creating an archive of the framework.



# Appendix B

## Safecode Patch

```

--- a/llvm/include/llvm/ADT/IntervalMap.h
+++ b/llvm/include/llvm/ADT/IntervalMap.h
@@ -1977,7 +1977,7 @@
     CurSize[Nodes] = CurSize[NewNode];
     Node[Nodes] = Node[NewNode];
     CurSize[NewNode] = 0;
-    Node[NewNode] = this->map->newNode<NodeT>();
+    Node[NewNode] = this->map->template newNode<NodeT>();
     ++Nodes;
 }

--- a/llvm/include/llvm/ADT/PointerUnion.h
+++ b/llvm/include/llvm/ADT/PointerUnion.h
@@ -263,7 +263,7 @@
     ::llvm::PointerUnionTypeSelector<PT1, T, IsInnerUnion,
     ::llvm::PointerUnionTypeSelector<PT2, T, IsInnerUnion, IsPT3 >
>::Return Ty;

-    return Ty(Val).is<T>();
+    return Ty(Val).template is<T>();
}

    /// get<T>() - Return the value of the specified pointer type. If the
@@ -276,7 +276,7 @@
     ::llvm::PointerUnionTypeSelector<PT1, T, IsInnerUnion,
     ::llvm::PointerUnionTypeSelector<PT2, T, IsInnerUnion, IsPT3 >
>::Return Ty;

-    return Ty(Val).get<T>();
+    return Ty(Val).template get<T>();
}

    /// dyn_cast<T>() - If the current value is of the specified pointer type,
--- a/llvm/tools/bugpoint/ToolRunner.cpp
+++ b/llvm/tools/bugpoint/ToolRunner.cpp
@@ -128,7 +128,7 @@
     ErrorFile.close();
}

-    errs() << OS;
+    errs() << OS.str();
}

    return ReturnCode;
--- a/llvm/projects/safecode/lib/InsertPoolChecks/RegisterBounds.cpp
+++ b/llvm/projects/safecode/lib/InsertPoolChecks/RegisterBounds.cpp
@@ -690,7 +690,7 @@
    Value * PH = ConstantPointerNull::get (getVoidPtrType(Context));
    Instruction * InsertBefore = &(F.getEntryBlock().front());
    RegisterVariableIntoPool(PH, &I, AllocSize, InsertBefore);
-    registeredArguments.push_back(std::make_pair<Value*, Argument*>(PH, &I));
+    registeredArguments.push_back(std::make_pair<Value*, Argument*>(&*PH, &*I));
}

}

--- a/llvm/projects/poolalloc/Makefile.common.in
+++ b/llvm/projects/poolalloc/Makefile.common.in
@@ -19,7 +19,7 @@
# Set the root directory of this project's install prefix
PROJINSTALLROOT := @prefix@

-CXXFLAGS += -Werror -Wall -Wno-deprecated
+CXXFLAGS += -Wall -Wno-deprecated

#
# Paths to various utilities
--- a/llvm/projects/safecode/runtime/SoftBoundRuntime/softboundcets.h
+++ b/llvm/projects/safecode/runtime/SoftBoundRuntime/softboundcets.h

```

```

@@ -241,7 +241,7 @@
#endif

-#define __WEAK_INLINE __attribute__((__weak__, __always_inline__))
+#define __WEAK_INLINE

    #if __WORDSIZE == 32
    #define __METADATA_INLINE __attribute__((__weak__))
--- a/llvm/projects/safecode/runtime/SoftBoundRuntime/softboundcets-wrappers.c
+++ b/llvm/projects/safecode/runtime/SoftBoundRuntime/softboundcets-wrappers.c
@@ -42,7 +42,7 @@
#include <arpa/inet.h>

    #if defined(__linux__)
-#include <bits/errno.h>
+#include <errno.h>
#include <sys/wait.h>
#include <wait.h>
#include <obstack.h>

```

# Appendix C

## Softbound Patch

```
--- a/softboundcets-34/softboundcets-lib/softboundcets-wrappers.c
+++ b/softboundcets-34/softboundcets-lib/softboundcets-wrappers.c
@@ -42,7 +42,7 @@
#include <arpa/inet.h>

#ifdef __linux__
#include <bits/errno.h>
#include <errno.h>
#include <sys/wait.h>
#include <wait.h>
#include <obstack.h>
--- a/softboundcets-34/softboundcets-lib/softboundmpx-wrappers.c
+++ b/softboundcets-34/softboundcets-lib/softboundmpx-wrappers.c
@@ -44,7 +44,7 @@
#include <arpa/inet.h>

#ifdef __linux__
#include <bits/errno.h>
#include <errno.h>
#include <sys/wait.h>
#include <wait.h>
#include <obstack.h>
```